ASSEMBLY CODE

https://xkcd.com/567/

LEARNING OBJECTIVES

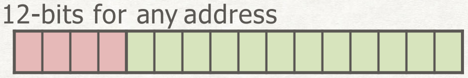
- The instructions of our machine in detail
- The difference between real and pseudo operations
- Desk checking assembly language programs
- Describe how an assembler translates assembly language programs into machine instructions

THE INSTRUCTIONS & THE MACHINE

- In this course you don't have to write assembly language programs, but you do have to be able to read and interpret them
- The names for opcodes are called mnemonics
- To do this you need to understand "exactly" what the instructions do
  - unfortunately the instruction set in the textbook isn't completely defined, so we will fill in the necessary information

OUR MACHINE

- Our machine has 16-bit memory cells or words (so our machine is not common, most machines have 8-bit (byte) cells, they are "byte-addressable")
- Also each instruction takes up one memory cell (16-bits), the opcode is 4-bits leaving 12-bits for any address



  - This means we have an address space of 2^{12} = 4096 cells or 8192 bytes - not a lot of memory
- The machine has a **single register R** for holding values during computation (there are other registers or latches we don't directly modify IR, PC, MAR, MDR, EQ, GT, LT)
- And our ADD and SUBTRACT instructions will use 2's complement 16 bit values (the textbook implies sign-magnitude, but that is silly - why?)

AND ALSO

- The contents of R stay the same until a new value is written in to it regardless of how many instructions have been executed since e.g. with a LOAD, ADD or SUBTRACT
- The contents of the condition bits LT, EQ and GT only get changed by the COMPARE instruction. They keep their values otherwise.

OUR INSTRUCTION SET

In lecture 12 we saw exactly what the processor does when executing some of these instructions: LOAD, STORE, ADD, JUMP, COMPARE, JUMPGT

FIGURE 6.5

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then GT = 1 else 0 if CON(X) = R then EQ = 1 else 0 if CON(X) < R then LT = 1 else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if GT = 1.
1010	JUMPEQ X	Get the next instruction from memory location X if EQ = 1.
1011	JUMPLT X	Get the next instruction from memory location X if LT = 1.
1100	JUMPNEQ X	Get the next instruction from memory location X if EQ = 0.
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

Typical assembly language instruction set

BRIEF DESCRIPTIONS

- X is the address (12 bits in machine code)
  - in assembly code it will be a textual label e.g. COUNT
  - we will see how to connect the address to the label shortly
- if R is on the left side of → it means the value stored there
- if it on the right side of → it means store the value in R

Instruction	Meaning
LOAD X	CON(X) → R
STORE X	R → CON(X)
CLEAR X	0 → CON(X)

BRIEF DESCRIPTIONS

- These are the available arithmetic operations
- We only have one register for arithmetic
  - this means we have to use direct access to memory for these instructions
- The increment and decrement instructions would be particularly expensive (and would require a hidden internal register)

Instruction	Meaning
ADD X	R + CON(X) → R
SUBTRACT X	R - CON(X) → R
INCREMENT X	CON(X) + 1 → CON(X)
DECREMENT X	CON(X) - 1 → CON(X)

- There are no increment or decrement R instructions, how would we do these operations?

BRIEF DESCRIPTIONS

- COMPARE is our most complicated instruction, we are comparing a value in memory with the current value in R
  - we set the condition codes appropriately
- The jump instructions depend on the values of the condition codes

Instruction	Meaning
COMPARE X	if CON(X) > R then 1 → GT else 0 if CON(X) = R then 1 → EQ else 0 if CON(X) < R then 1 → LT else 0
JUMPGT X	if GT = 1 then X → PC
JUMPEQ X	if EQ = 1 then X → PC
JUMPLT X	if LT = 1 then X → PC
JUMPNEQ X	if EQ = 0 then X → PC
JUMP X	X → PC

BRIEF DESCRIPTIONS

- We will assume there is some magic with the IN and OUT instructions
  - IN takes a decimal integer and stores its 16-bit 2's complement value in the memory location
  - OUT takes the value at the memory location and displays its signed decimal value
- HALT stops the computer from executing any more instructions

Instruction	Meaning
IN X	number typed → CON(X)
OUT X	CON(X) → display
HALT	stop execution

PSEUDO-OPERATIONS

- **Pseudo-op**: commands in the program directed to the assembler, not converted to machine instructions:
  - .BEGIN and .END to mark where program is
  - .DATA to mark memory location as holding data:

```
COUNTER:    .DATA  0
X:          .DATA  12
```

  - this is how we connect the label to its address (the assembler works out that the address is as we shall see)

STRUCTURE OF A PROGRAM

FIGURE 6.6

```
.BEGIN      --This must be the first line of the program
  ⋮         --Assembly language instructions like those in Figure 6.5
    HALT    --This instruction terminates execution of the program
  ⋮         --Data generation pseudo-ops such as
            --.DATA are placed here, after the HALT
.END        --This must be the last line of the program
```

Structure of a typical assembly language program

EXAMPLES

Simple examples:

x = y + 3

```
             LOAD Y
             ADD THREE
             STORE X
…            -- Data comes after HALT
X:           .DATA 0 — X is initially 0
Y:           .DATA 5 — Y is initially 5
THREE:       .DATA 3 — The constant 3
```

input a and b

while a > b do

print a

a = a – 2

```
             IN A
             IN B
LOOP1:       LOAD B
             COMPARE A
             JUMPLT LOOP1END
             JUMPEQ LOOP1END
             OUT A
             LOAD A
             SUBTRACT TWO
             STORE A
             JUMP LOOP1
LOOP1END: …  -- Data comes after HALT
A:           .DATA 0
B:           .DATA 0
TWO:         .DATA 2
```


ADDING UP NUMBERS

FIGURE 6.7

Step	Operation
1	Set the value of *Sum* to 0
2	Input the first number *N*
3	While *N* is not negative do
4	Add the value of *N* to *Sum*
5	Input the next data value *N*
6	End of the loop
7	Print out *Sum*
8	Stop

Algorithm to compute the sum of nonnegative numbers

ASSEMBLERS AND ASSEMBLY LANGUAGE (12 OF 20)

FIGURE 6.8

```
        .BEGIN                 --This marks the start of the program
        CLEAR       SUM        --Set the running sum to 0 (line 1)
        IN          N          --Input the first number N (line 2)
--The next three instructions test whether N is a negative number (line 3)
AGAIN:  LOAD        ZERO       --Put 0 into register R
        COMPARE     N          --Compare N and 0
        JUMPLT      NEG        --Go to NEG if N < 0
--We get here if N ≥ 0. We add N to the running sum (line 4)
        LOAD        SUM        --Put SUM into R
        ADD         N          --Add N. R now holds (N + SUM)
        STORE       SUM        --Put the result back into SUM
--Get the next input value (line 5)
        IN          N
--Now go back and repeat the loop (line 6)
        JUMP        AGAIN
--We get to this section of the program only when we encounter a negative value
NEG:    OUT         SUM        --Print the sum (line 7)
        HALT                   --and stop (line 8)
--Here are the data generation pseudo-ops
SUM:    .DATA       0          --The running sum goes here
N:      .DATA       0          --The input data are placed here
ZERO:   .DATA       0          --The constant 0
--Now we mark the end of the entire program
        .END
```

Assembly language program to compute the sum of nonnegative numbers

DESK CHECK

- Stepping through the code as if we were the computer is known as desk checking
- We keep track of the variables - R and memory values
- e.g.

```
         LOAD        A
LOOP:    COMPARE     TWO
         JUMPGT      FINISH
         SUBTRACT    TWO
         JUMP        LOOP
FINISH:  STORE       A
         OUT         A
         HALT
A:       .DATA       4
TWO:     .DATA       2
```

R	?
A:	4
TWO:	2
OUTPUT	

HOW ASSEMBLERS WORK

Translation and Loading

- **Assembler** translates to machine language:
  - Converts symbolic op codes to binary equivalents
  - Converts symbolic labels to memory addresses
  - Performs pseudo-op actions
  - Writes **object file** containing machine instructions
- **Loader** gets program ready to run:
  - Places instructions in memory
  - Triggers the hardware to run the program

MNEMONICS TO OPCODES

Converting symbolic op codes to binary

- Assembler maintains a table
- Assembler looks up symbolic op codes in the table and substitutes the binary analogue
- Use binary search to optimize table lookups
  - better to use a hash table (CS130)

FIGURE 6.9

Operation	Binary Value
ADD	0011
CLEAR	0010
COMPARE	0111
DECREMENT	0110
HALT	1111
OUT	1110
⋮	
STORE	0001
SUBTRACT	0101

Structure of the op code table

LABELS AND SYMBOL TABLE

Converting symbolic labels to memory addresses

- Assembler needs two **passes**
  - Looks over assembly code two times
- First pass:
  - Keeps a count of how many instructions from the start
  - Collects symbolic labels and adds to **symbol table** along with location counter

SYMBOL TABLE

FIGURE 6.10

Label	Code		Location Counter
LOOP:	IN	X	0
	IN	Y	1
	LOAD	X	2
	COMPARE	Y	3
	JUMPGT	DONE	4
	OUT	X	5
	JUMP	LOOP	6
DONE:	OUT	Y	7
	HALT		8
X:	.DATA	0	9
Y:	.DATA	0	10

(a)

Symbol Table

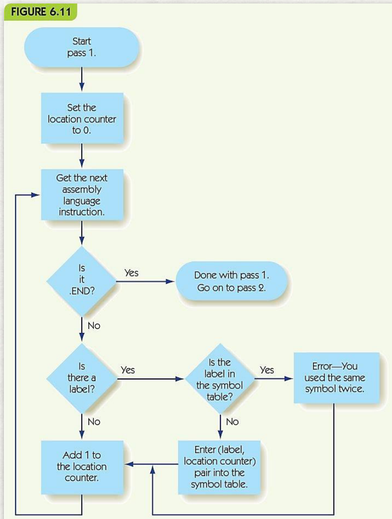
Symbol	Address Value
LOOP	0
DONE	7
X	9
Y	10

(b)

Generation of the symbol table

ASSEMBLER FIRST PASS ALGORITHM

FIGURE 6.11



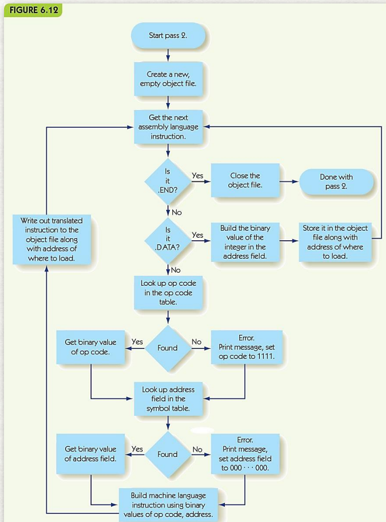
Outline of pass 1 of the assembler

SECOND PASS

- Second pass:
  - Looks up and replace op codes
  - Substitutes label references with location from symbol table
  - Sets up .DATA pseudo-ops with location and binary value
  - Writes instructions to object file

ASSEMBLER SECOND PASS ALGORITHM

FIGURE 6.12



Outline of pass 2 of the assembler

OBJECT FILE

FIGURE 6.13

Instruction Format: **Op Code** 4 bits **Address** 12 bits

Object Program:

Address	*Machine Language Instruction*	*Meaning*
0000	1101 000000001001	IN X
0001	1101 000000001010	IN Y
0010	0000 000000001001	LOAD X
0011	0111 000000001010	COMPARE Y
0100	1001 000000000111	JUMPGT DONE
0101	1110 000000001001	OUT X
0110	1000 000000000000	JUMP LOOP
0111	1110 000000001010	OUT Y
1000	1111 000000000000	HALT
1001	0000 000000000000	The constant 0
1010	0000 000000000000	The constant 0

Example of an object program

SUMMARY

- System software creates a virtual environment that is easy for users to use.
- Assemblers and loaders are system software: translate human-friendly programs to machine language.
- Assembly language uses symbolic names, symbolic op codes, and pseudo-ops to describe algorithms.
- Assembler translates source programs to object files; loader places object instructions in memory.